

UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

*Lab 3: Embarrassingly parallelism with
OpenMP: Mandelbrot set*

*Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110*



10th April 2019, Q2

Contents

1	Introduction	2
2	Task decomposition analysis with <i>Tareador</i>	3
2.1	Point decomposition	3
2.2	Row decomposition	4
2.3	Characteristics of the TDG	6
3	Point decomposition in OpenMP	7
3.1	OpenMP Task Implementation	7
3.1.1	OpenMP Taskwait Variant	10
3.1.1.1	The Taskwait Clause	13
3.1.2	OpenMP Taskgroup Variant	14
3.2	OpenMP Taskloop Implementation	16
3.2.1	The Nogroup Clause	17
4	Row decomposition in OpenMP	21
4.1	OpenMP For Implementation	24
5	Conclusion	28
6	Annex	29
6.1	Scalability plots of point strategy using taskloop implementation . . .	29
6.2	Scalability plots of row strategy using taskloop implementation . . .	34

1 Introduction

Nowadays, the aim of parallelizing codes is to improve the performance of nearly all the applications that can be computed using classical computer architectures. Nevertheless, there are several ways to parallelize the same application, some better than others.

In our case, we are going to study different tasking models in OpenMP to parallelize the Mandelbrot set, a particular set of point, in the complex domain, whose boundary generates a distinctive and easily recognisable two-dimensional fractal shape¹.

Based on the maximum number of steps the code does to calculate wheater a point belongs to the set or not, the coloring of the image changes. The points that belong to the set reach the maximum limit of steps. In the figure 1, those points are the ones in black color. Hence, the points which are not black are the ones that do not belong to the set.

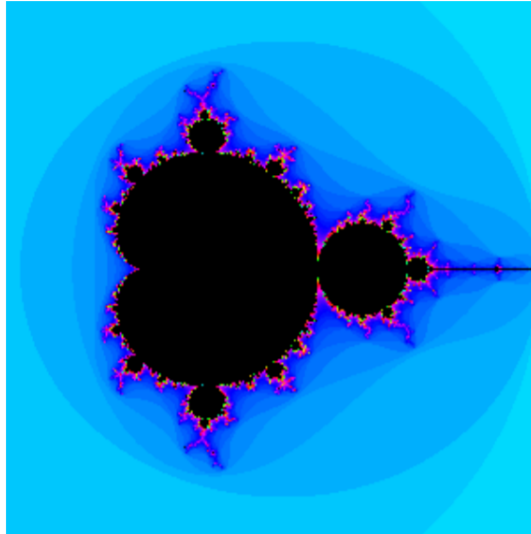


Figure 1: Fractal shape.

First of all we are going to see with *Tareador* the binaries which are potentially parallelizable. Afterwards, we are going to explore two parallelization methods: Point and Row decomposition. To do it, we are going to use the following tasking models: explicit task implementation (task only, task with `taskwait/taskgroup/task-loop`) and implicit task implementation (for-based implementation). Finally, we are going to compare the best implementation of one method with the best implementation of the other one.

¹To know more about the Mnadelbrot set, it is highly recommended to look at the following link: http://en.wikipedia.org/wiki/Mandelbrot_set

2 Task decomposition analysis with *Tareador*

In this section, we had to analyse the two possible task granularities that could be exploited in the given program. To do it, we used the *Tareador* tool, which was very useful to see graphically the created tasks and which dependences exist between them.

2.1 Point decomposition

In this decomposition strategy, a task corresponds with the computation of a single point (row, col) of the Mandelbrot set. Thus, we will have $row \times col$ tasks.

In order to analyse the potential parallelism of this strategy, we modified the code in *mandel-tar.c* to create several *Tareador* tasks. As we are using a point decomposition strategy, the tasks are created inside the most inner loop of the function *mandelbrot*. Figure 2 shows the fragment of the function that we modified.

```
for (row = 0; row < height; ++row) {  
    for (col = 0; col < width; ++col) {  
        tareador_start_task("point");  
  
        ...  
  
        tareador_end_task("point");  
    }  
}
```

Figure 2: Modified fragment of the *mandel-tar.c* code.

Once we did this, we executed interactively *mandel-tar* and *mandel-tar* using *run-tareador.sh* script. The first one is used for timing purposes and to check for the numerical validity of the output (-o option) whereas with the second one we can visualize the Mandelbrot set.

The script has defined inside the size of the image to compute (-w option). In this case, the value was 8 to generate a reasonable task graph in a reasonable execution time. Hence, as we said before, the number of tasks will be $8 \times 8 = 64$.

Figures 3 and 4 below show the two task decomposition graphs (TDG) of the two possible executions. There are some nodes bigger than the others. This is because these nodes have executed more instructions than the rest. Therefore, these nodes represent pixels in white².

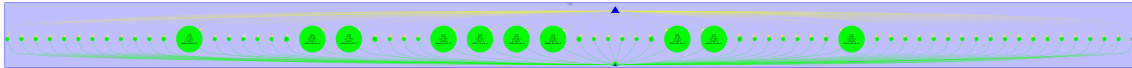


Figure 3: Mandel-tar task decomposition graph using the point decomposition strategy.

²The bigger the node, the more iterations it does in the do while fragment of the function. This make that the computed color is white.

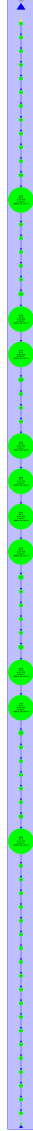


Figure 4: Mandel-tar task decomposition graph using the point decomposition strategy.

This strategy will have more overhead of creation and termination of tasks than in the row strategy because it has to create more tasks. However, as a positive point the tasks are better distributed because if in a row there are many white areas, this work will not only be done by a single thread. Thus, it may end the execution earlier than in the row decomposition strategy.

2.2 Row decomposition

In this other strategy, a task corresponds with the computation of a whole row of the Mandelbrot set. This strategy only creates *row* tasks.

In this case, the code is not the same as before. We changed the creation of the *Tareador* tasks so that each time we enter a new row (second loop) a new task is created. The modified version of the code is shown below.

```

for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
        ...
    }
    tareador_end_task("row");
}

```

Figure 5: Modified fragment of the *mandel-tar.c* code.

Afterwards, we executed interactively *mandel-tar* and *mandeld-tar* again. We used the same size as before. Hence, the total number of tasks will be 8. The graphical results we got can be seen in figures 6 and 7.

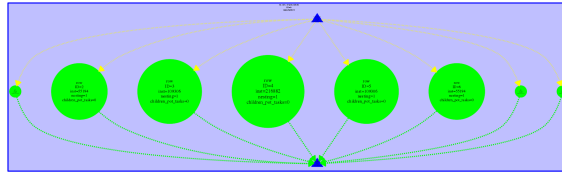


Figure 6: Mandel-tar task decomposition graph using the row decomposition strategy.



Figure 7: Mandeld-tar task decomposition graph using the row decomposition strategy.

In this strategy, the overhead time of creation and termination of tasks will be very small in comparison with the point strategy because it has to create less tasks. Nevertheless, it may happen that in a full row, all of its pixels must be white. In this case, it can happen that while other threads have finished their work, this other thread is still executing the row. Consequently, it is possible that the execution time may be bigger than in the point decomposition strategy.

2.3 Characteristics of the TDG

We saw in the previous sections that the execution of the mandel-tar has a very different task dependence graph than the execution of the mandeld-tar.

On the one hand, we can see that in mandel-tar every point is independent from the others. Consequently, we could parallelize that fragment of the code using OpenMP clauses.

On the other hand, in mandeld-tar we can see that all iterations have become sequential. In this situation, we do not gain anything by parallelizing the code, but we increase the execution time because of the overhead of creation and termination of tasks.

Using the *Tareador* we could see which variable was responsible of creating those dependences. We did the following: Right Click into a task -> Data View -> Edges-out -> Real Dependency. Figure 8 shows the result we got.

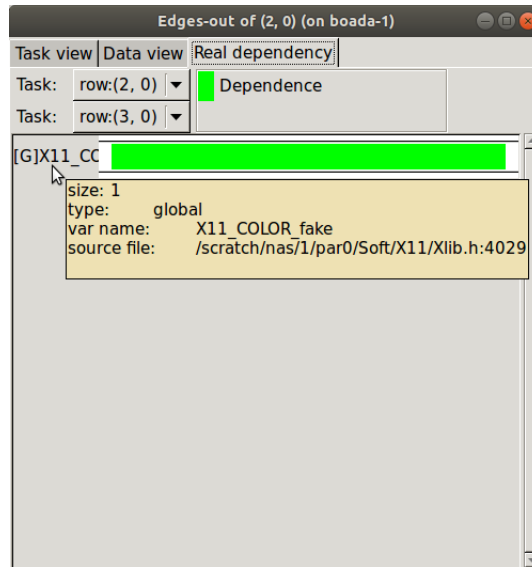


Figure 8: Variable that provokes the dependences between tasks.

We can see that there is only one variable that is causing all the dependences: `X11_COLOR_fake`. Observing the code, we noticed that the only difference between mandel-tar and mandeld-tar was a fragment of the code that was only executed in mode `_DISPLAY_` (mandeld-tar):

```

#if _DISPLAY_
/* Scale color and display point */
long color = (long) ((k-1) * scale_color) + min_color;
if (setup_return == EXIT_SUCCESS) {
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
#else
    output[row][col]=k;
#endif

```

Figure 9: Fragment of the *mandel-tar.c* code.

Therefore, variable `X11_COLOR_fake` is used at least in one of the functions `XSetForeground` and `XDrawPoint`. We could protect this section of code in the parallel OpenMP code using the *critical* clause to define a region of mutual exclusion where only one thread can be working at the same time.

```

#if _DISPLAY_
/* Scale color and display point */
long color = (long) ((k-1) * scale_color) + min_color;

#pragma omp critical
if (setup_return == EXIT_SUCCESS) {
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
#else
    output[row][col]=k;
#endif

```

Figure 10: Fragment of the *mandel-tar.c* code using the *critical* clause to protect a fragment of the code.

3 Point decomposition in OpenMP

In this section, we are going to explore different options in the OpenMP tasking model to express the Point decomposition for the Mandelbrot computation program. We will analyse the scalability and behaviour of these options.

3.1 OpenMP Task Implementation

The aim of this tasking model strategy is to create a task for each point. Figure 11 show the OpenMP clauses we used to implement the task strategy (the code can be found in file *mandel-omp-task-point.c* in codes folder).


```

for (int row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp single
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col)
        {
            ...
            #pragma omp critical
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
            }
            ...
        }
    }
}

```

Figure 11: Fragment of the *mandel-omp-task-point.c* code showing the OpenMP clauses to implement the task strategy.

For every row, only one thread is creating all the tasks and inserting them into a pool of tasks, while the other threads are taking tasks from the pool and executing them. Nevertheless, the "role" of creating tasks changes for every row. Thus, each iteration of the col loop will be executed as an independent task. The `firstprivate` clause is used in order to avoid problems of data races³. Finally, the `critical` clause is used to honour the dependences we detected for the graphical version in the previous section. However, this section of the code will not be called since we will not execute the graphical version (only *mandel-tar*).

The following figure shows the execution flow of the program using the *Paraver* tool.

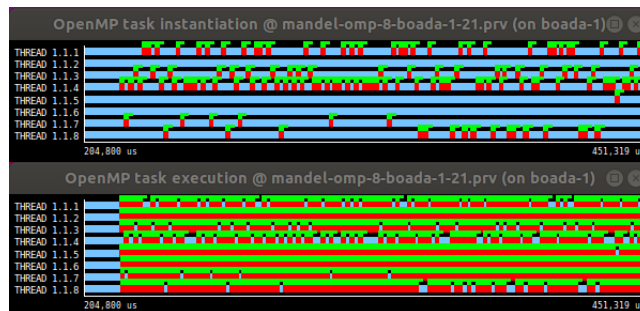
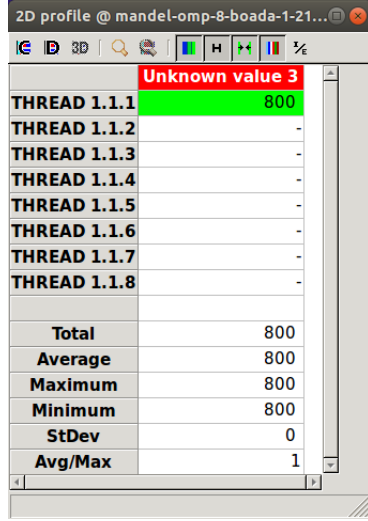


Figure 12: Zoomed part of the execution flow using the task strategy.

We can observe the effect of the `single` clause. In the upper part (task instantiation) only one thread is creating the tasks and inserting them in the task pool. In the bottom part (task execution) we can see that the other threads execute the tasks as they are put in the pool. The role of the task creator changes every time we finish a row.

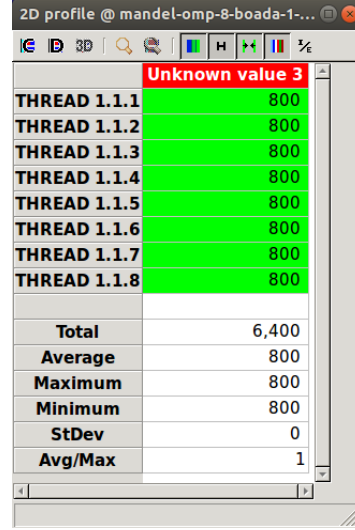
³We have not used the `private` clause because it initializes the variable with a random value, and we wanted the real value of the variable (iteration).

In total, there are 640000 tasks created and executed. This is because the image has size 800×800 . We have seen this using the "OMP_parallel_functions.cfg" and "OMP_state_profile.cfg" configurations (figure 15). The parallel construct is executed 800 times (figure 13), the number of rows. Finally, the single worksharing construct is executed 800 times for each thread (figure 14). Thus, the single clause is executed 6400 times ($rows \times threads$). All threads will execute the single line but only one will "gain". This thread will be responsible for creating the tasks of that row.



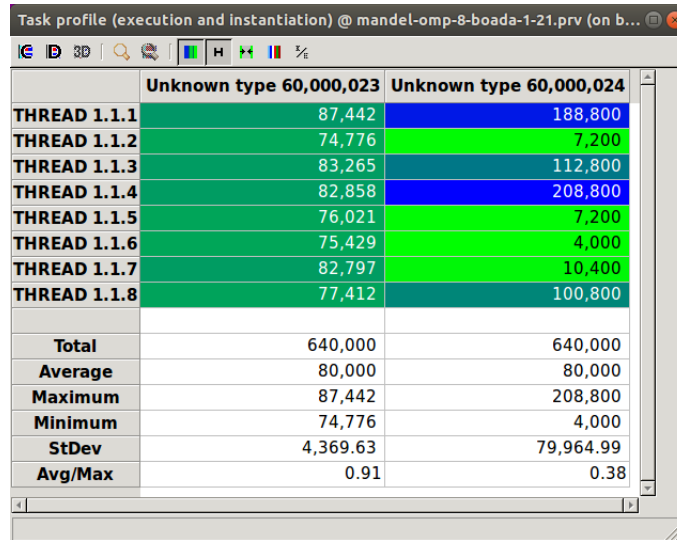
Unknown value 3	
THREAD 1.1.1	800
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	800
Average	800
Maximum	800
Minimum	800
StDev	0
Avg/Max	1

Figure 13: Table with the total number of times `#pragma omp parallel` has been called by each thread.



Unknown value 3	
THREAD 1.1.1	800
THREAD 1.1.2	800
THREAD 1.1.3	800
THREAD 1.1.4	800
THREAD 1.1.5	800
THREAD 1.1.6	800
THREAD 1.1.7	800
THREAD 1.1.8	800
Total	6,400
Average	800
Maximum	800
Minimum	800
StDev	0
Avg/Max	1

Figure 14: Table with the total number of times `#pragma omp single` has been called by each thread.



	Unknown type 60,000,023	Unknown type 60,000,024
THREAD 1.1.1	87,442	188,800
THREAD 1.1.2	74,776	7,200
THREAD 1.1.3	83,265	112,800
THREAD 1.1.4	82,858	208,800
THREAD 1.1.5	76,021	7,200
THREAD 1.1.6	75,429	4,000
THREAD 1.1.7	82,797	10,400
THREAD 1.1.8	77,412	100,800
Total	640,000	640,000
Average	80,000	80,000
Maximum	87,442	208,800
Minimum	74,776	4,000
StDev	4,369.63	79,964.99
Avg/Max	0.91	0.38

Figure 15: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

Finally, figures 16 and 17 show the time and speedup plots.

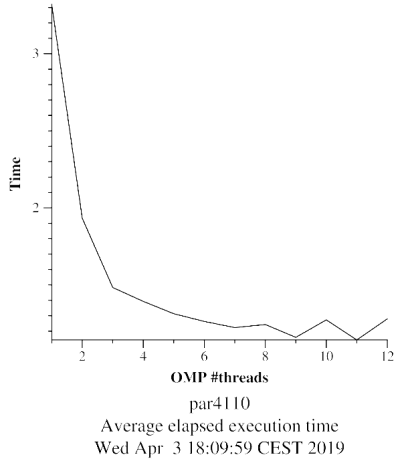


Figure 16: Execution time plot varying the number of threads.

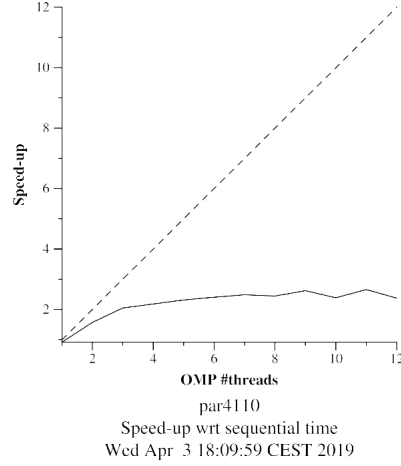


Figure 17: Speedup plot varying the number of threads.

In the first plot, the execution time is reduced as we increase the number of threads until the time stays constant. However, we think that if we increase a lot the number of threads, the execution time would end up increasing because of the overheads.

On the other hand, in the second plot the speedup increases a lot with 2-4 threads but ends up constant as we increase the number of threads.

Finally, the scalability is not appropriate because the speedup stays at 2 when we are using a big number of threads.

3.1.1 OpenMP Taskwait Variant

In this variant of the previous code, only one thread (the one that gets access to the single region), traverses all iterations of the row and col loops, generating a task for each iteration of the innermost loop (point). To do it, we have introduced "`#pragma omp taskwait`" at the end of each iteration of a row. Consequently, the creator thread must wait until all tasks for a row finish. After that, the thread will advance one iteration of the row loop and generate a new bunch of tasks. This new version of the code can be found in *mandel-omp-task-point-taskwait.c*.

```

#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
            ...
        }
    }
    #pragma omp taskwait // waiting point for all child tasks
}

```

Figure 18: Fragment of the *mandel-omp-task-point-taskwait.c* code showing the OpenMP clauses to implement the task strategy with the taskwait variant.

We can clearly see in figures 19, 20 and 23 that the only thread that is creating tasks is the thread 0. Moreover, in the zoomed figure we can observe that every time it creates some tasks, then it has to wait a little bit until it can create tasks again. This is because of the taskwait clause we added at the end of the outermost loop. Thread 0 will create all the tasks for a unique row and will wait until all these tasks terminate.

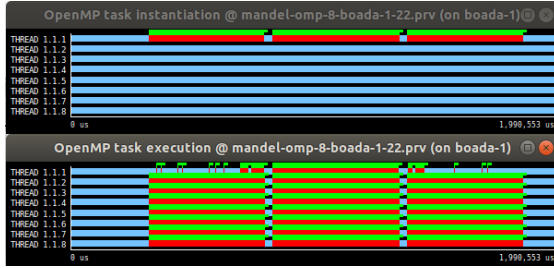


Figure 19: Execution flow using the taskwait strategy.

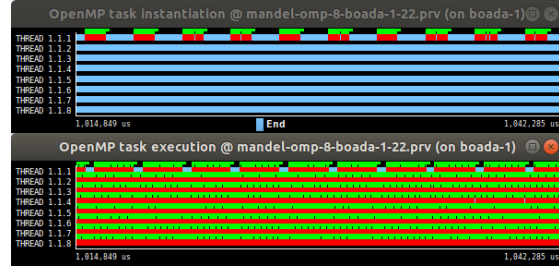


Figure 20: Zoomed part of Figure 19.

The number of created and executed tasks is still 640000 as we have not modified the size of the image (figure 23). Nevertheless, the number of calls to parallel is 1 (figure 21), and the number of calls to the single worksharing construct is now 8 (figure 22), the number of threads. Besides, the taskwait clause will be executed 800 times (number of rows). The granularity is the same than before. Each task has exactly one iteration of the innermost for loop.

REGION (open)	
THREAD 1.1.1	1
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	1
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 21: Table with the total number of times `#pragma omp parallel` has been called by each thread.

92 (mandel-omp.c, mandel-omp)	
THREAD 1.1.1	1
THREAD 1.1.2	1
THREAD 1.1.3	1
THREAD 1.1.4	1
THREAD 1.1.5	1
THREAD 1.1.6	1
THREAD 1.1.7	1
THREAD 1.1.8	1
Total	8
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 22: Table with the total number of times `#pragma omp single` has been called by each thread.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,224	640,000
THREAD 1.1.2	70,094	-
THREAD 1.1.3	79,978	-
THREAD 1.1.4	84,120	-
THREAD 1.1.5	73,685	-
THREAD 1.1.6	74,480	-
THREAD 1.1.7	84,387	-
THREAD 1.1.8	84,032	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,224	640,000
Minimum	70,094	640,000
StDev	6,185.05	0
Avg/Max	0.90	1

Figure 23: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

From here, as we will not change the `#pragma omp parallel` and `#pragma omp single` anymore, figures 21 and 22 will be the same for the following variants and new implementations.

Figures 24 and 25 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the previous section.

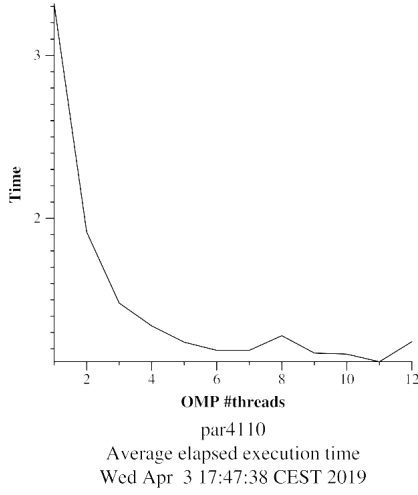


Figure 24: Execution time plot varying the number of threads.

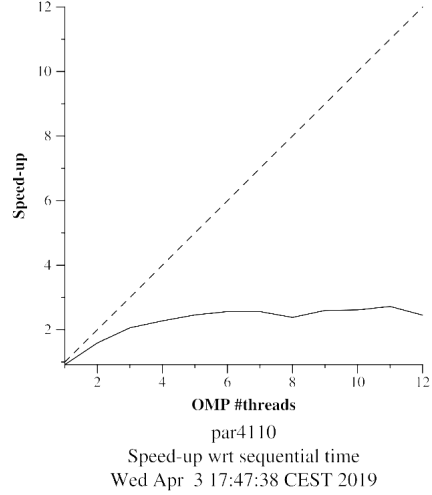


Figure 25: Speedup plot varying the number of threads.

3.1.1.1 The Taskwait Clause

In reality, the taskwait clause is not necessary because tasks do not have any dependence between them. We saw in section 2 that the mandel-tar binary does not produce any kind of dependence between tasks. Thus, we can create and execute all tasks without having to wait until some of them terminate. The modified version of the code can be found in the file *mandel-omp-task-point-taskwait-without.c*.

In order to see the differences with respect to the version with the taskwait clause, we have repeated the evaluation of scalability and tracing. The results are shown below.

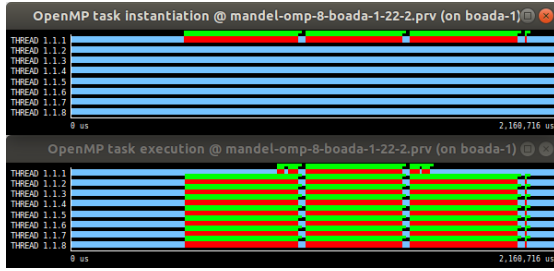


Figure 26: Execution flow using the taskwait strategy without the barrier.

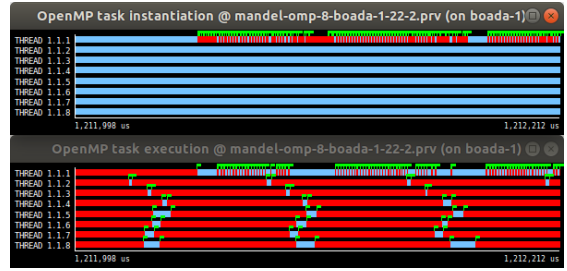


Figure 27: Zoomed part of Figure 26.

The number of tasks created/executed has not changed because we have not touched the task clause or the parallel and single clauses. However, as we have deleted the taskwait clause, now the creator thread (thread 0 in our case) does not have to wait until the tasks of the same row terminate. Consequently, it can create all possible tasks of the program at the same time. Nevertheless, we see in figures 26 and 27 that this is not happening as expected.

We though that thread 0 would create all possible tasks and then execute the remaining tasks inside the task pool. What is happening is that this thread creates some tasks, then stops the task generation, then executes some taks, and then

proceeds generating new tasks. After reasoning about this, we have concluded that the task pool that OpenMP has is limited, it has a maximum number of threads inside the pool. As a consequence, when thread 0 reaches this limit it has to stop creating tasks, so it begins to execute the created tasks until the total number of tasks inside the pool is reduced. Afterwards, it begins to create tasks again.

On the other hand, we can see that the scalability plots (time and speedup) have improved a bit even though it is not a significative change.

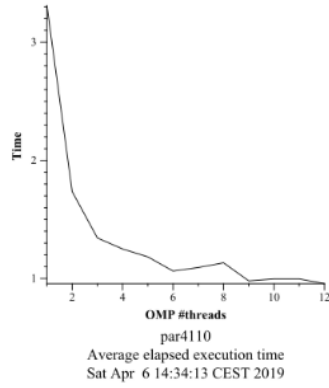


Figure 28: Execution time plot varying the number of threads.

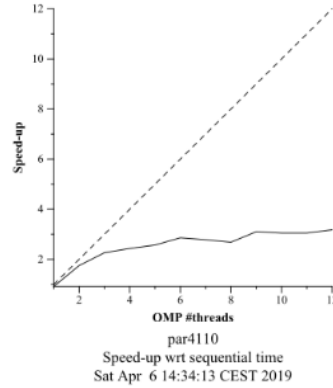


Figure 29: Speedup plot varying the number of threads.

To sum up, we can delete the `taskwait` clause because it is not necessary for `mandel-tar`. This change improves a little bit the performance of the execution. Moreover, without this clause the creator thread has to stop generating tasks because it reaches the maximum limit of the pool. Thus, it will be changing from creating to executing and vice versa very often.

3.1.2 OpenMP Taskgroup Variant

In this other variant, we define a region in the program where the thread will wait for the termination of all descendant (not only child) tasks. Figure 30 shows a section of the modified code that can be found in file `mandel-omp-task-point-taskgroup.c` inside the codes directory. Now, the tasks of the same row are generated without any specific order. However, the creator thread has to wait until all the tasks of the same row terminate.

```

#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    {
        for (int col = 0; col < width; ++col) {
            #pragma omp task firstprivate(row, col)
            {
                ...
            }
        }
    }
}

```

Figure 30: Fragment of the *mandel-omp-task-point-taskgroup.c* code showing the OpenMP clauses to implement the task strategy with the taskgroup variant.

In the execution flow we obtained with *Paraver* we can see that there is no difference at all with respect to the execution flow of the previous section. This is because in reality the taskgroup clause does not have descendants of its childs. As a consequence, it does the same effect than using the taskwait clause.

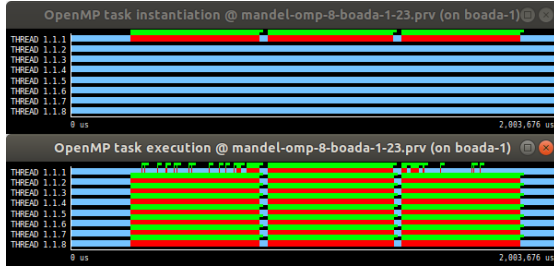


Figure 31: Execution flow using the taskwait strategy without the barrier.

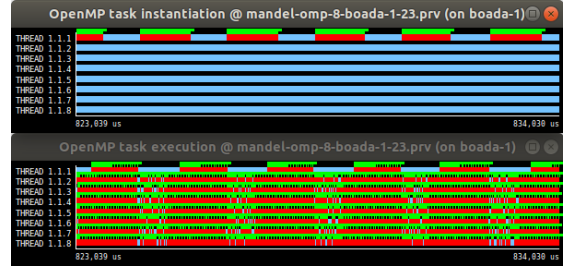


Figure 32: Zoomed part of Figure 31.

The number of created tasks is the same (640000). Moreover, the number of calls to parallel and the single worksharing construct is the same (1 and 8 respectively). The only difference is that in this variant the taskgroup clause is executed 800 times (number of rows).

Task profile (execution and instantiation) @ mandel-omp-8-boada-1-23.prv (on boada-1)		
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,180	640,000
THREAD 1.1.2	79,923	-
THREAD 1.1.3	78,184	-
THREAD 1.1.4	76,028	-
THREAD 1.1.5	81,165	-
THREAD 1.1.6	80,066	-
THREAD 1.1.7	75,837	-
THREAD 1.1.8	80,617	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,180	640,000
Minimum	75,837	640,000
StDev	3,623.98	0
Avg/Max	0.91	1

Figure 33: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

Figures 34 and 35 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the two previous sections.

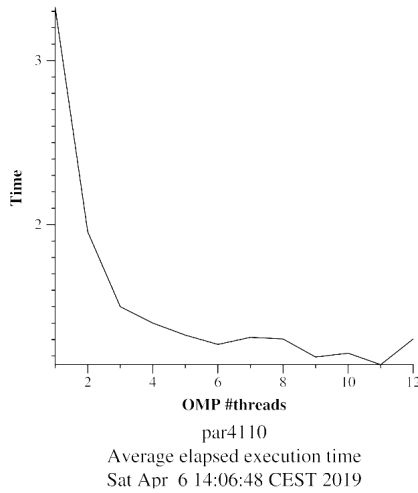


Figure 34: Execution time plot varying the number of threads.

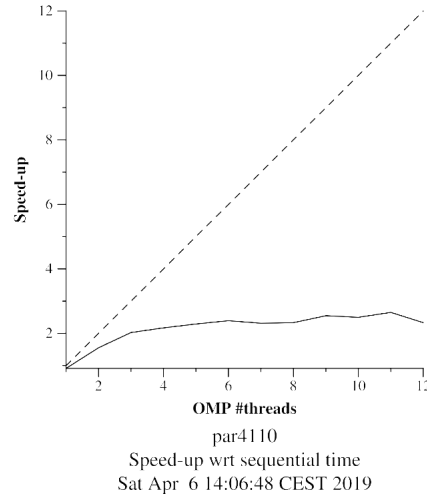


Figure 35: Speedup plot varying the number of threads.

3.2 OpenMP Taskloop Implementation

In this new strategy, the taskloop clause is used. It generates tasks out of its iterations of a for loop, allowing to better control the number of tasks generated or their granularity (num_tasks or grainsize). This version of the code can be found in the file *mandel-omp-task-point-taskloop-800*.

```

#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(800)
    for (int col = 0; col < width; ++col) {
        ...
    }
}

```

Figure 36: Fragment of the *mandel-omp-task-point-taskloop-800.c* code showing the OpenMP clauses to implement the task strategy with the taskloop variant.

If we create 800 tasks of the mostinner loop, in reality we are executing an equivalent version of the second task version we analysed. In order to analyse the performance of this strategy we will use the traces generated with *Paraver* and using the appropriate configuration files.

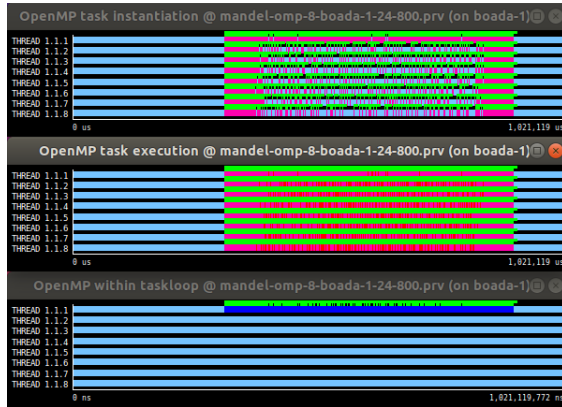


Figure 37: Execution flow of using the taskloop strategy and 800 tasks for each row.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	181,716	7,407
THREAD 1.1.2	69,573	806
THREAD 1.1.3	72,344	812
THREAD 1.1.4	64,350	747
THREAD 1.1.5	67,880	767
THREAD 1.1.6	63,368	731
THREAD 1.1.7	64,957	733
THREAD 1.1.8	67,812	797
Total	652,000	12,800
Average	81,500	1,600
Maximum	181,716	7,407
Minimum	63,368	731
StDev	37,978.22	2,195.05
Avg/Max	0.45	0.22

Figure 38: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

First of all, we can see something strange in figure 37 in the instantiation part. In the first moment all threads create a lot of tasks, then only thread 0 is the one creating most of the tasks, and then all threads create again all the tasks. The reasoning about this effect will be done in the next section, when comparing the results with another 2 traces.

On the other hand, in figure 38 we can see that there are created more than 640000 tasks, the expected number. The reason we got when asking to the professor was that when using the *Paraver* and the taskloop, there are created some metatasks to evaluate some things. Thus, the number of created tasks is increased a bit.

3.2.1 The Nogroup Clause

The taskloop construct has an implicit task barrier (taskgroup) at the end. This is causing the effect of a taskwait clause. As we reasoned before about that we could

delete the taskwait clause because there are no dependences between tasks, we can do the same with the taskgroup clause. To do it, the taskloop construct accepts a nogroup clause that eliminates this implicit barrier.

Now, we will explore how this new version behaves in terms of performance when using 8 threads and for different task granularities (800, 400, 200, 100, 50, 25, 10, 5, 2 and 1 tasks). All versions of the codes can be found in files *mandel-omp-task-point-taskloop- $\{granularity\}$ -nogroup*.

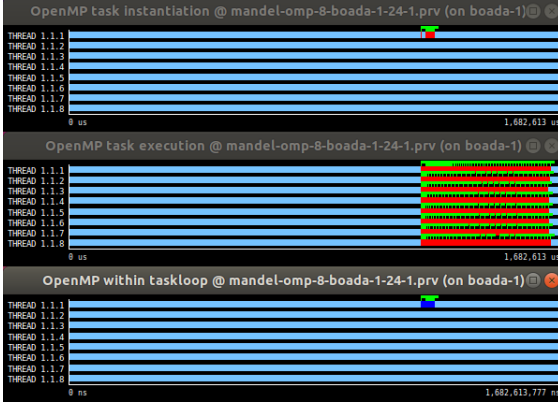


Figure 39: Execution flow of using the taskloop strategy and 1 task for each row.



Figure 40: Execution flow of using the taskloop strategy and 50 tasks for each row.

In the previous traces we can observe multiple things. Firstly, the time expended creating tasks is reduced as we increase the granularity. It is common sense, if we create less tasks per row, we will expend less time doing it (thanks to the nogroup clause). Moreover, the execution time of the parallel part is also reduced with a smaller number of tasks. We think that this happen because there will be less overhead of creation, synchronization and termination and while some tasks execute the longer tasks, the rest will continue executing the other ones.

Secondly, we can see in figure 37 a different behaviour with respect to figures 39 and 40. All threads are creating tasks whereas in the other 2 traces only thread 0 is the one creating tasks.

Our conclusion is that in the version with number of tasks 800 threads end faster the upper, left and right parts of the non-created image because there are no white parts. Hence, as we have the nogroup clause and there are no more tasks inside the pool, they can create tasks too. However, in the middle of the image, as there are more white pixels, threads expend more time executing the task, so in this part they will create less tasks and thread 0 will be responsible for creating the biggest part of tasks. As we decrease the total number of tasks, threads will do more instructions per task. Thus, they will not have time to create tasks and only thread 0 will be responsible for creating all the tasks (from number of tasks 100 down, only thread 0 is creating the tasks).

The following plots are the scalability plots using the same implementation but varying the number of tasks per row. They are only a part of all. The rest of the plots can be found in section 6.1.

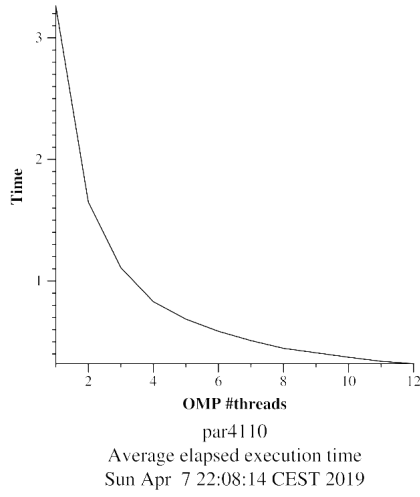


Figure 41: Time plot varying the number of threads and using 1 task.

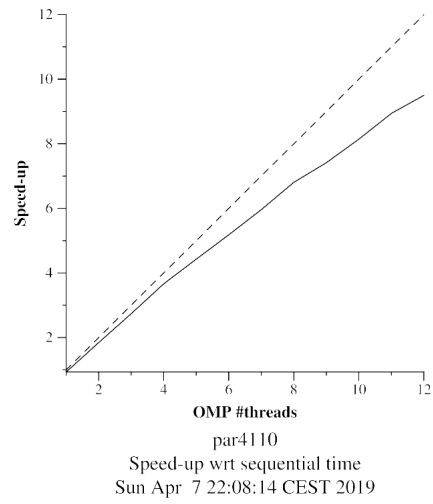


Figure 42: Speedup plot varying the number of threads and using 1 task.

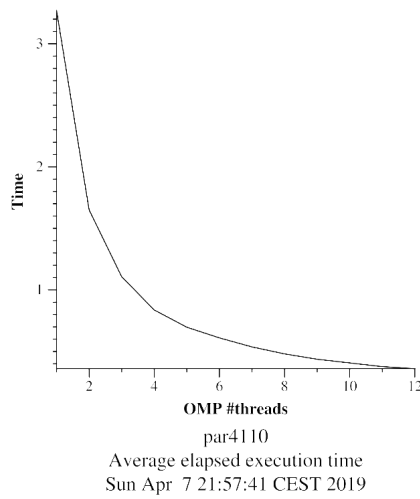


Figure 43: Time plot varying the number of threads and using 50 tasks.

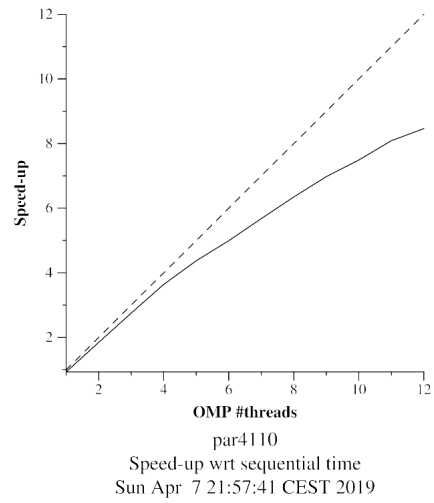


Figure 44: Speedup plot varying the number of threads and using 50 tasks.

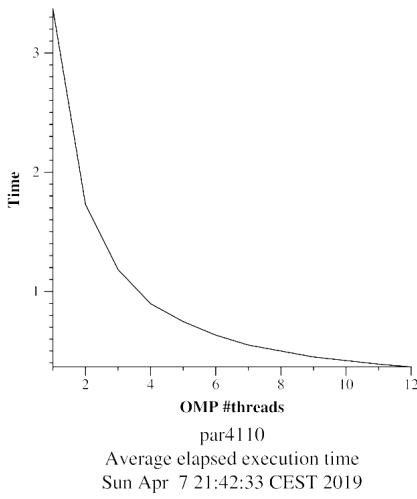


Figure 45: Time plot varying the number of threads and using 800 tasks.

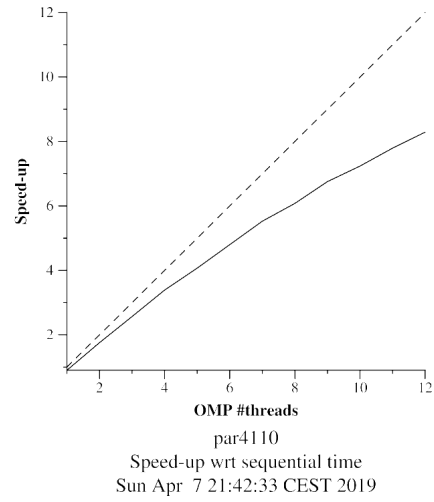


Figure 46: Speedup plot varying the number of threads and using 800 tasks.

In the previous plots we can see the tendency of the execution time and speedup as we increase the number of tasks. The bigger the number of tasks, the worse the execution time and speedup. This is because of the overhead of creation, synchronization and termination of tasks. Creating one task per pixel is not a good idea.

Besides, these results are much better than the ones of the previous section, so it seems that this implementation is the best for a point strategy.

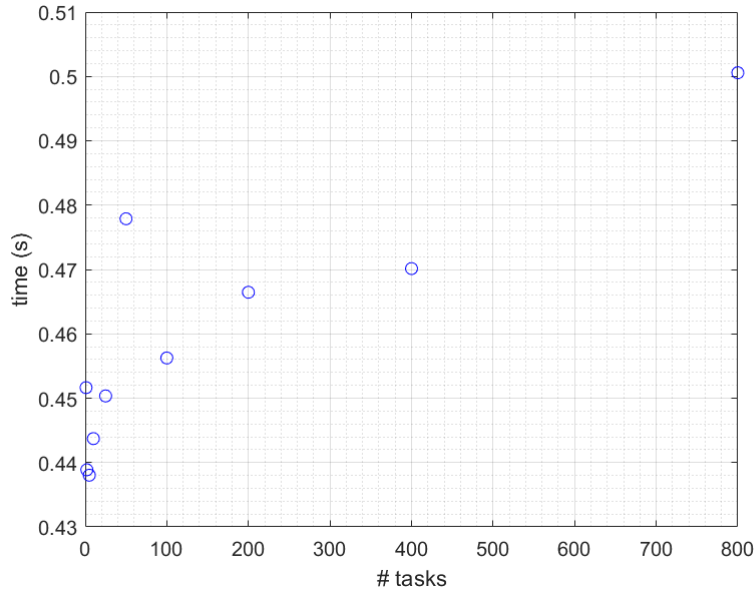


Figure 47: Execution time plot when varying the number of tasks (number of threads = 8).

Figure 47 shows the execution time of the program with a constant number of threads (8). The bigger the granularity (less number of tasks), the lower the

execution time (and the lower speedup). The main reason is that we have to create less tasks. Thus, the overhead of creation and termination of tasks is smaller. The best option for this strategy is using small values for the number of tasks per row. In this situation, only 1 or 2 threads will execute the full row, while the other ones can execute the following rows.

4 Row decomposition in OpenMP

In this section, we are going to analyse the performance of the Row decomposition strategy using the best implementation we got with the Point strategy. Afterwards, we will compare both results. All versions of the codes can be found in files *mandel-omp-task-row-taskloop-{granularity}-nogroup*.

The best implementation in the previous section was using the taskloop clause. Thus, the new code is the following:

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop firstprivate(row) num_tasks(800) nogroup
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        ...
    }
}
```

Figure 48: Fragment of the *mandel-omp-task-row-taskloop-800-nogroup.c* code showing the OpenMP clauses to implement the task strategy with the taskloop variant.

The following plots are the scalability plots using the same implementation but varying the number of tasks per row. They are only a part of all. The rest of the plots can be found in section 6.2.

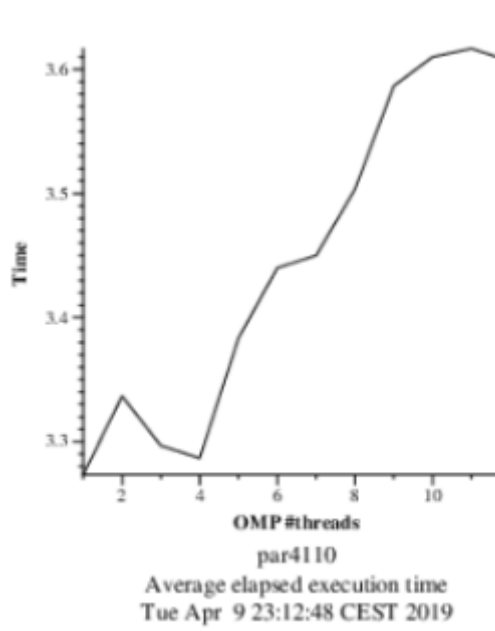


Figure 49: Execution time plot varying the number of threads (Num. tasks = 1).

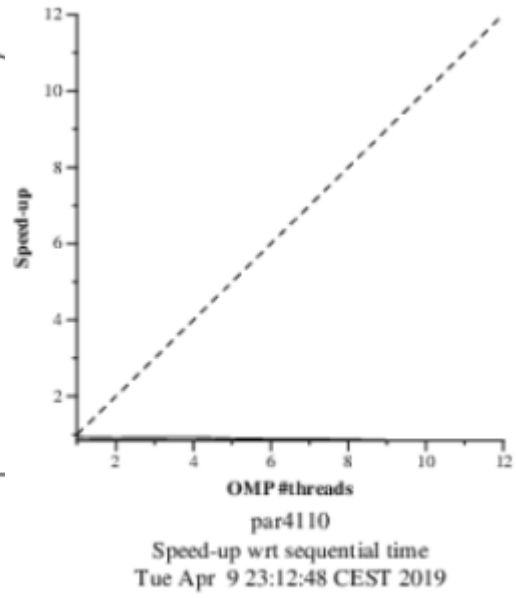


Figure 50: Speedup plot varying the number of threads (Num. tasks = 1).

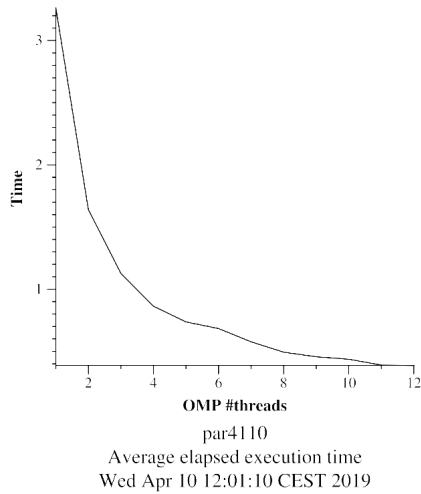


Figure 51: Execution time plot varying the number of threads (Num. tasks = 50).

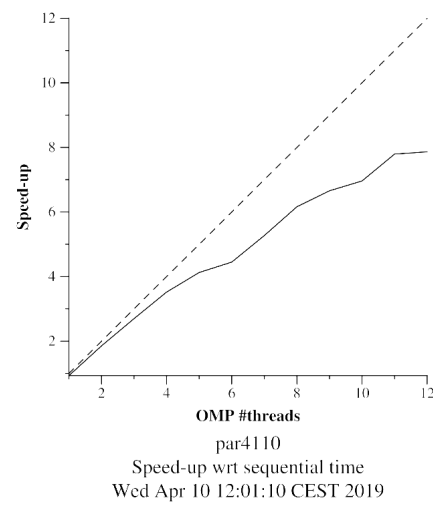


Figure 52: Speedup plot varying the number of threads (Num. tasks = 50).

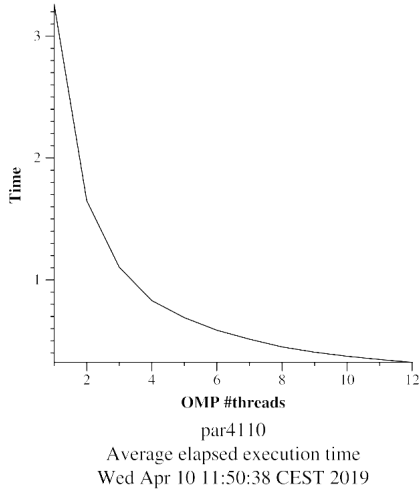


Figure 53: Execution time plot varying the number of threads (Num. tasks = 800).

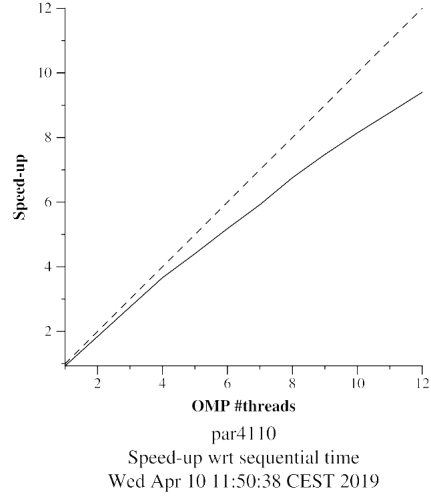


Figure 54: Speedup plot varying the number of threads (Num. tasks = 800).

We can see that now the tendency is the other way round with respect to the previous section. The bigger the number of tasks, the better the execution time. Now, the important thing is not the overhead but the distribution of the tasks into the different threads. Thus, a big number of tasks will balance better the work of the threads.

Now, in figure 55 we can see that the opposite happens. The bigger the granularity, the bigger the execution time. Thus, in the row strategy, the best option is to create as much tasks as possible. The main reason is that in this strategy, what we are modifying is the number of rows each thread executes. Hence, the bigger the number of tasks, the lower granularity (less instructions). This means that every thread will do less work per thread and will end faster, will take another task and so on.

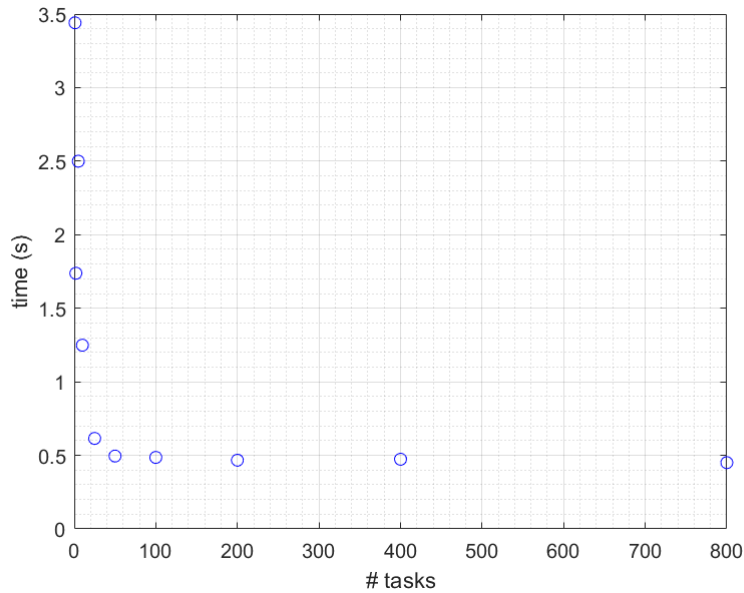


Figure 55: Execution time plot when varying the number of tasks (number of threads = 8).

4.1 OpenMP For Implementation

In this section, we are going to implement the for-based parallelization. With this new strategy, instead of tasking to distribute the work among the threads, we distribute loop iterations within the team of threads that encounters this work-sharing construct. The different codes can be found in the *for* directory inside *codes* directory.

```
#pragma omp parallel for schedule(static) private(col)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        ...
    }
}
```

Figure 56: Fragment of the code *mandel-omp-for-static.c*.

The following plots show the execution time and speedup for all possible schedules we know. We thought that a value of 10 iterations per chunk would very accurate for a good solution. With a big number of iterations per chunk, as we are in the row strategy, one thread would have done a lot of work. Thus, the results would have been very bad.

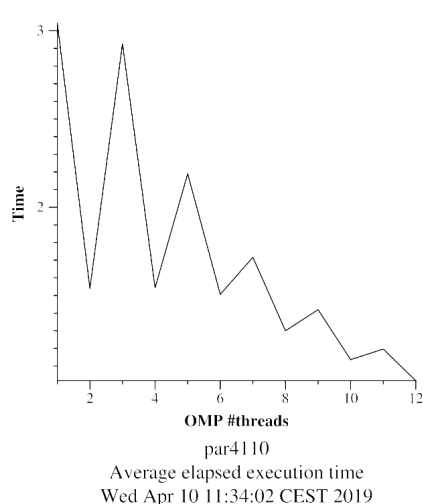


Figure 57: Execution time plot varying the number of threads (schedule(static)).

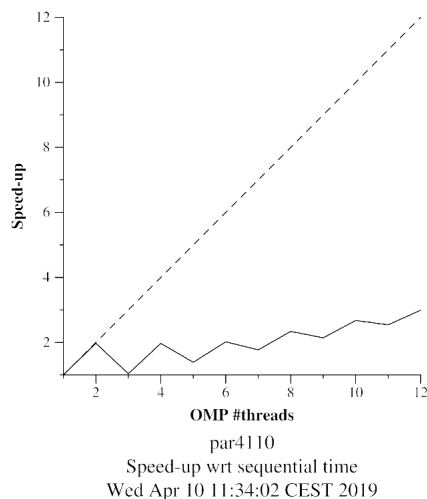


Figure 58: Speedup plot varying the number of threads (schedule(static)).

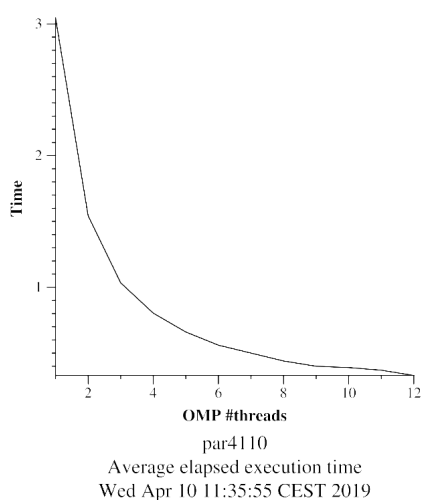


Figure 59: Execution time plot varying the number of threads (schedule(static, 10)).

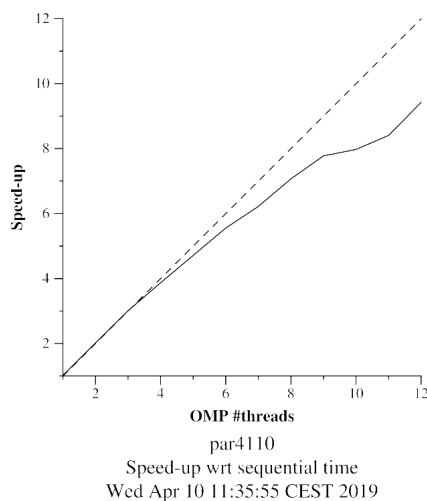


Figure 60: Speedup plot varying the number of threads (schedule(static, 10)).

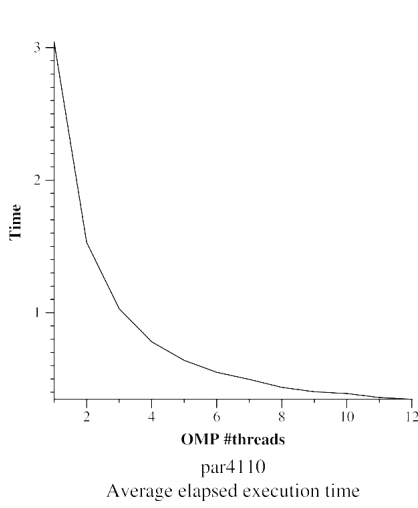


Figure 61: Execution time plot varying the number of threads (schedule(dynamic, 10)).

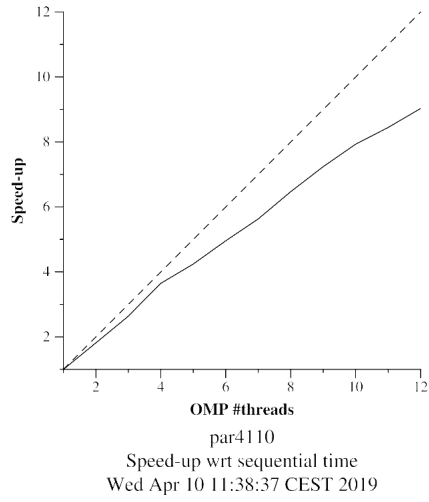


Figure 62: Speedup plot varying the number of threads (schedule(dynamic, 10)).

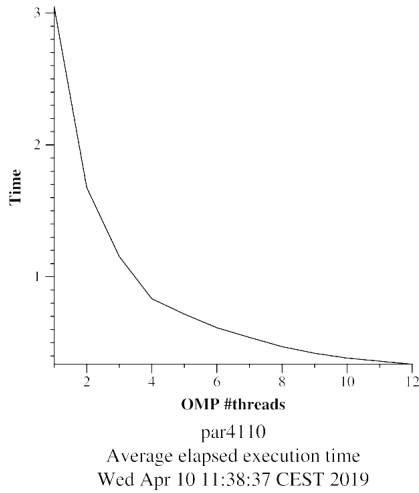


Figure 63: Execution time plot varying the number of threads (schedule(guided, 10)).

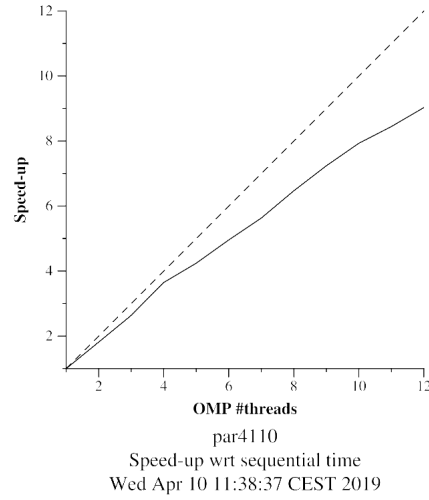


Figure 64: Speedup plot varying the number of threads (schedule(guided, 10)).

In figure 65, we can observe the different time executions when using only 8 threads. It is obvious that only static would be the one with a greater time execution because threads that have the tasks in the middle of the picture have to do more iterations per pixel, as it is where most of the white pixels are located. However, if we use 10 iterations per chunk with the static schedule, the time is reduced a lot because now, all threads will execute tasks that have white pixels, so they will end faster.

On the other hand, the dynamic strategy is the best option because whenever one thread ends, OpenMP will give him another task of the same chunk size without losing time.

Finally, the guided strategy has not given the expected results. It is slower than `schedule(static, 10)`. The reason is that with this strategy, we give a big number of iteration to each thread, until we arrive to the chunk size limit. Thus, some threads will have to execute tasks with a big number of white pixels.

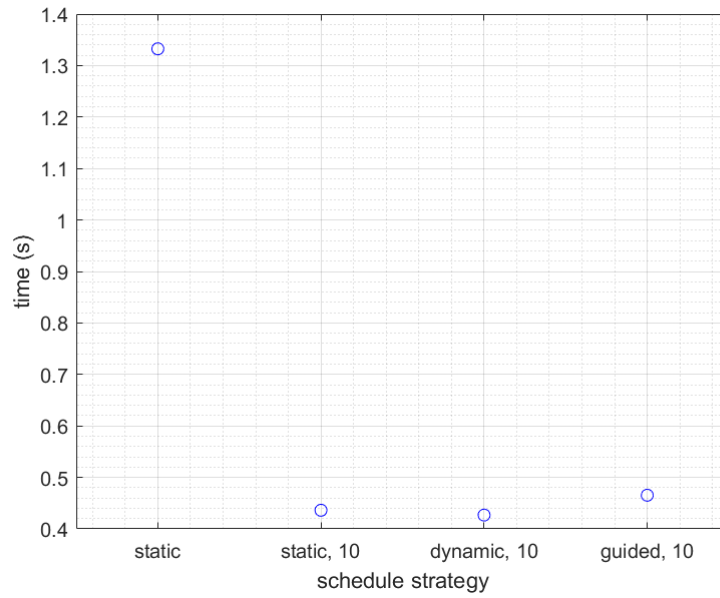


Figure 65: Execution time plot for the different schedule methods.

5 Conclusion

In these 3 laboratory sessions we have seen different strategies and tasking methods on 2 nested loops.

We have found that the innermost loop iterations have different execution times because some points belong to the Mandelbrot set and others do not. Thus, some of them have to reach the maximum number of steps.

This has led us to find that the best results in the Point strategy are obtained when tasks get a few number of iterations of the innermost loop.

On the other hand, the excessive creation of tasks leads us to an increment of the execution times due to that each task has a creation, synchronization and termination times.

Using different methods of creation of explicit tasks we have observed that the best method is using the taskloop clause because it leads us to control in an easier way the granularity of each task.

Moreover, we have seen that the best granularity for one strategy (for example, Point) is at the same time the worst granularity for the other strategy, and vice versa. The reason is because one strategy needs to create a lot of tasks to improve its parallelism (Row strategy) whereas the other one has to limit its creation of tasks due to the overhead (Point strategy).

Finally, we have seen that using implicit creation task in the Row strategy, the best option to use is the `schedule(dynamic, 10)` because it is the one that distributes better all the work.

6 Annex

6.1 Scalability plots of point strategy using taskloop implementation

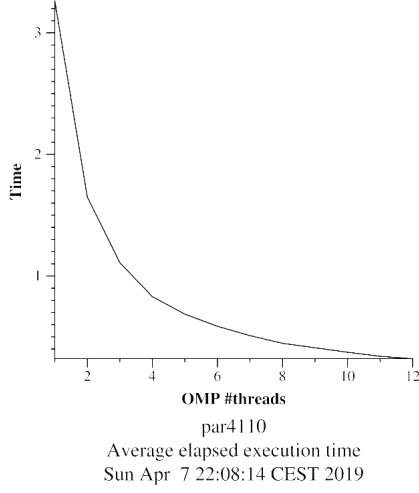


Figure 66: Execution time plot varying the number of threads (Num. tasks = 1).

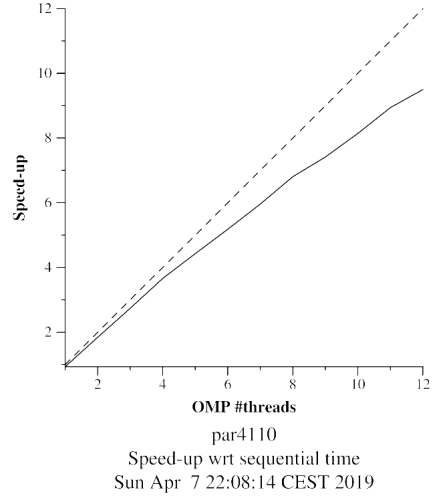


Figure 67: Speedup plot varying the number of threads (Num. tasks = 1).

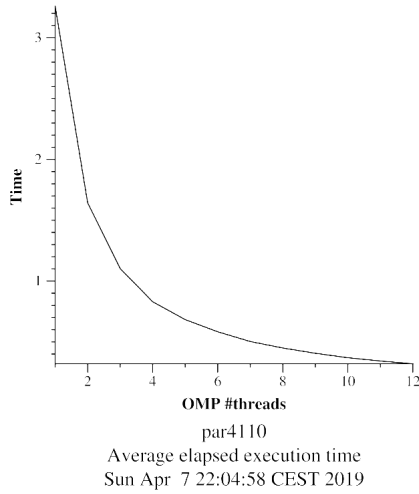


Figure 68: Execution time plot varying the number of threads (Num. tasks = 2).

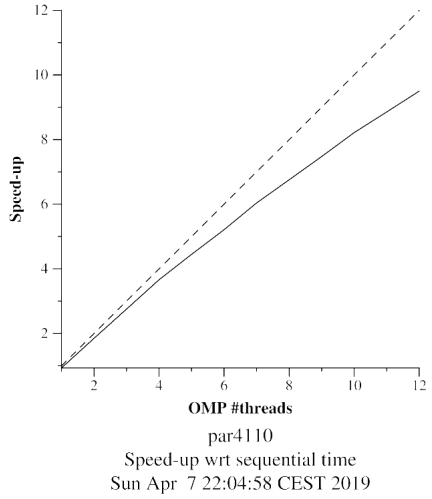


Figure 69: Speedup plot varying the number of threads (Num. tasks = 2).

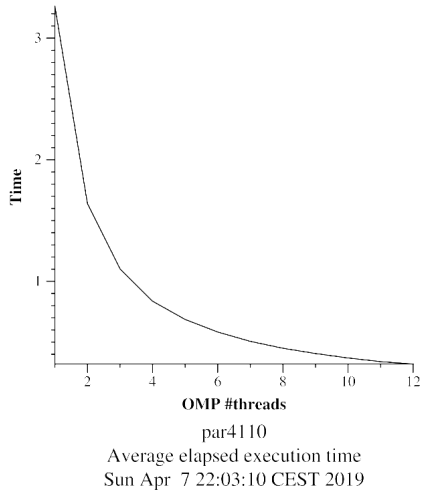


Figure 70: Execution time plot varying the number of threads (Num. tasks = 5).

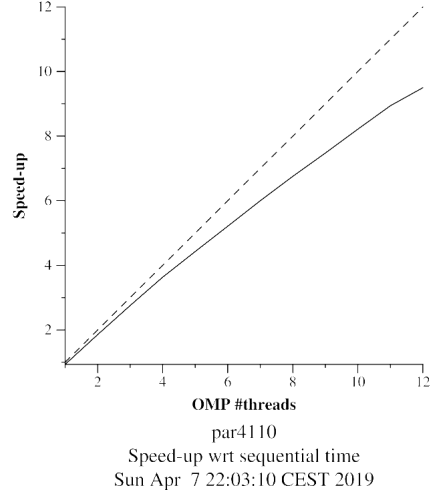


Figure 71: Speedup plot varying the number of threads (Num. tasks = 5).

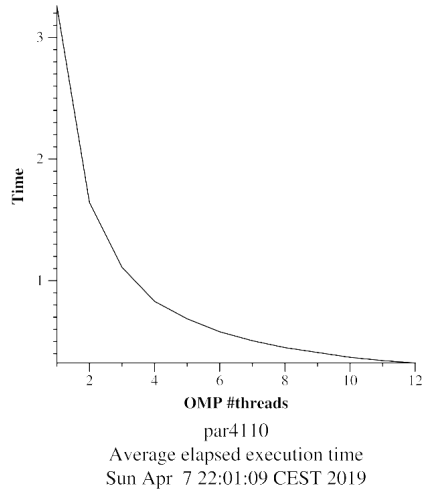


Figure 72: Execution time plot varying the number of threads (Num. tasks = 10).

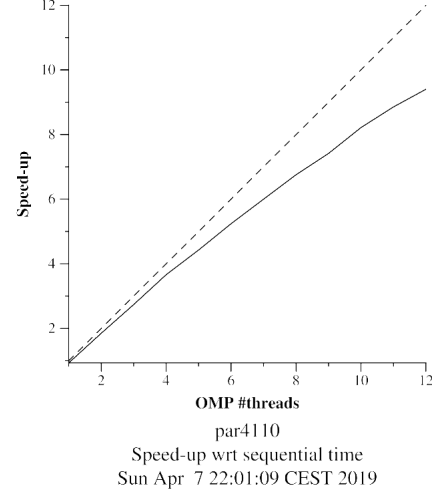


Figure 73: Speedup plot varying the number of threads (Num. tasks = 10).

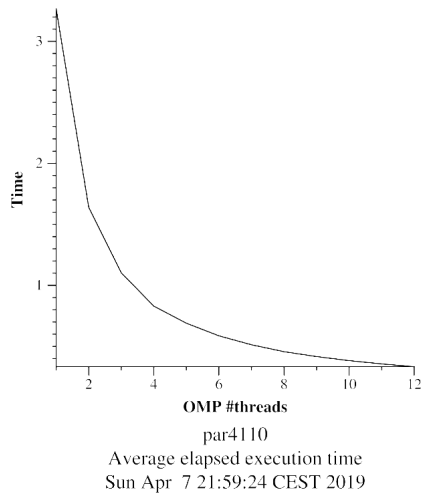


Figure 74: Execution time plot varying the number of threads (Num. tasks = 25).

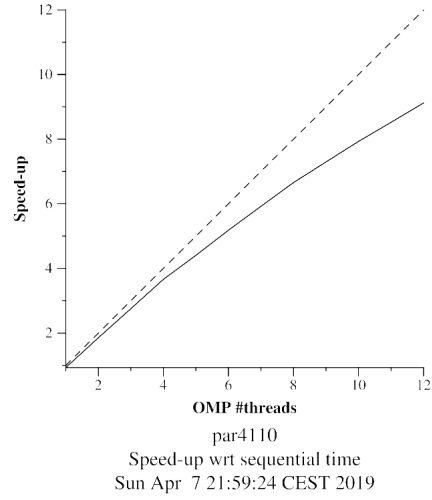


Figure 75: Speedup plot varying the number of threads (Num. tasks = 25).

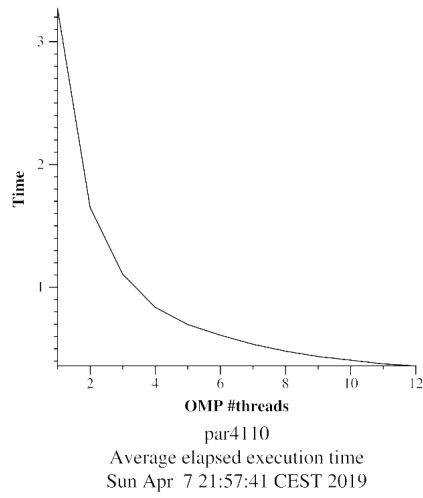


Figure 76: Execution time plot varying the number of threads (Num. tasks = 50).

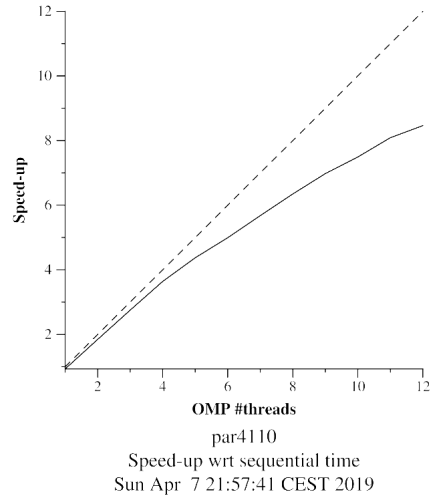


Figure 77: Speedup plot varying the number of threads (Num. tasks = 50).

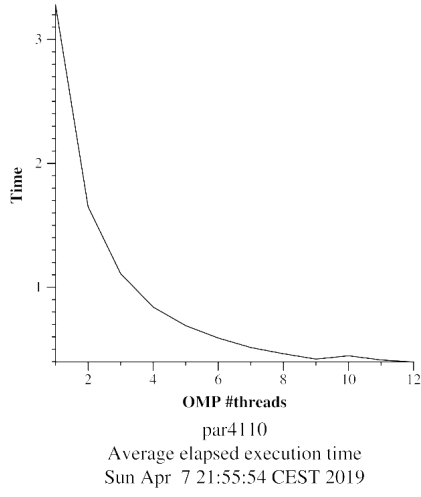


Figure 78: Execution time plot varying the number of threads (Num. tasks = 100).

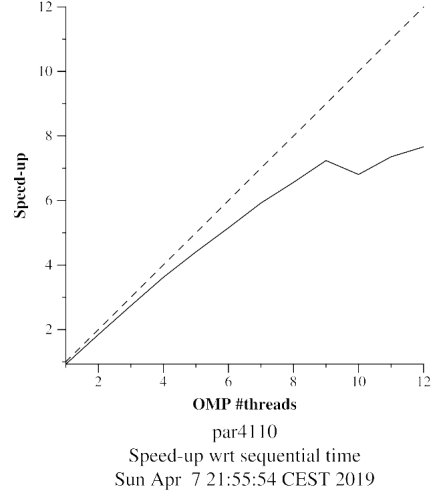


Figure 79: Speedup plot varying the number of threads (Num. tasks = 100).

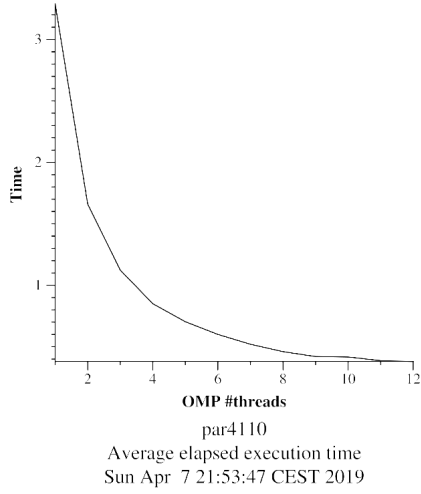


Figure 80: Execution time plot varying the number of threads (Num. tasks = 200).

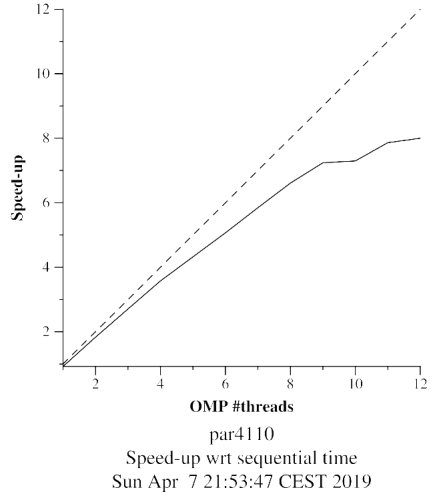


Figure 81: Speedup plot varying the number of threads (Num. tasks = 200).

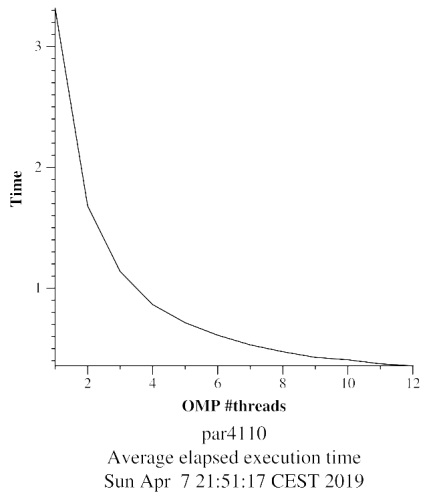


Figure 82: Execution time plot varying the number of threads (Num. tasks = 400).

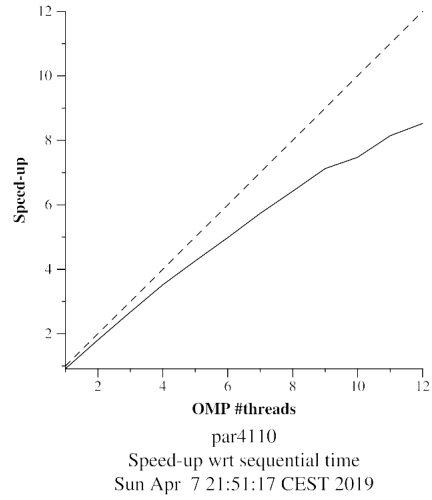


Figure 83: Speedup plot varying the number of threads (Num. tasks = 400).

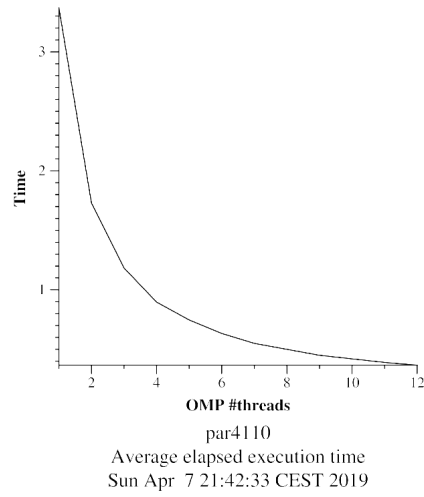


Figure 84: Execution time plot varying the number of threads (Num. tasks = 800).

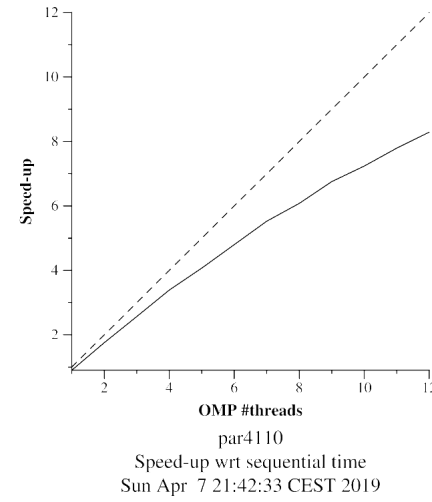


Figure 85: Speedup plot varying the number of threads (Num. tasks = 800).

6.2 Scalability plots of row strategy using taskloop implementation

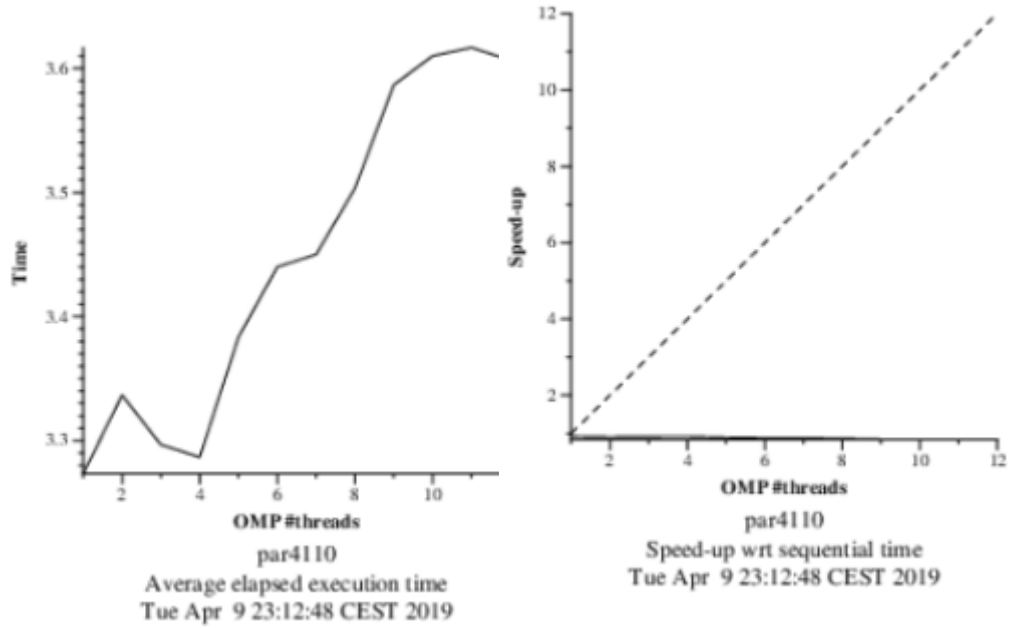


Figure 86: Execution time plot varying the number of threads (Num. tasks = 1).

Figure 87: Speedup plot varying the number of threads (Num. tasks = 1).

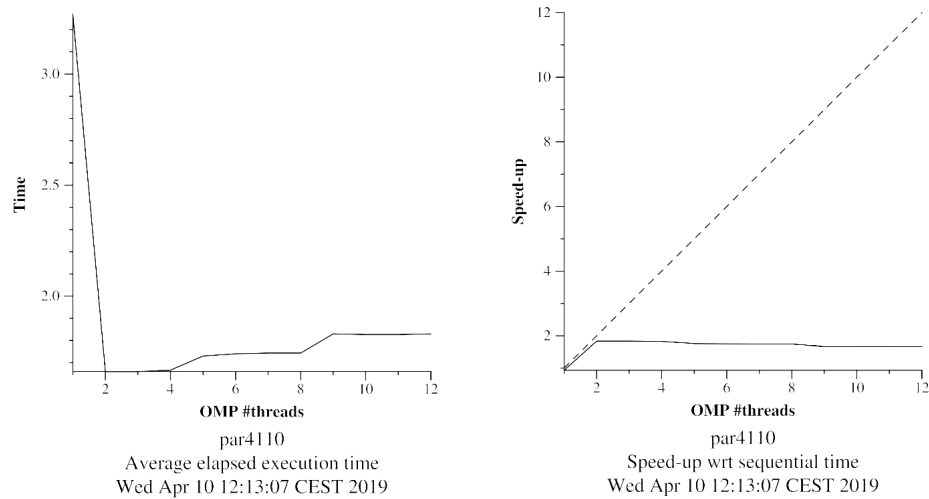


Figure 88: Execution time plot varying the number of threads (Num. tasks = 2).

Figure 89: Speedup plot varying the number of threads (Num. tasks = 2).

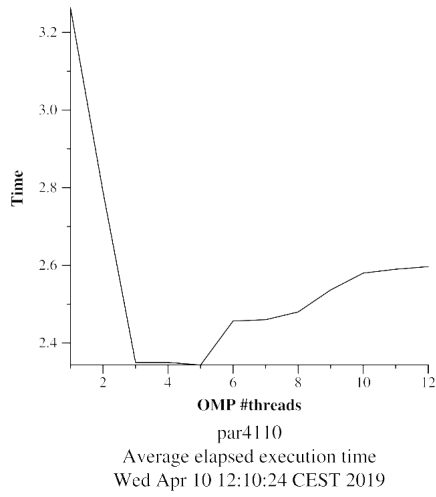


Figure 90: Execution time plot varying the number of threads (Num. tasks = 5).

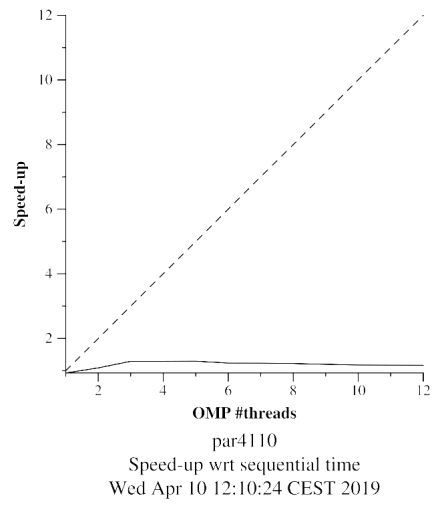


Figure 91: Speedup plot varying the number of threads (Num. tasks = 5).

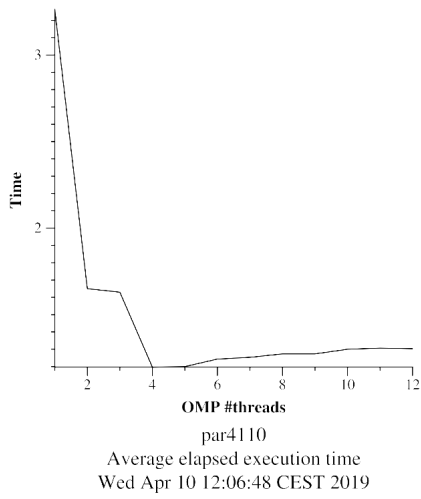


Figure 92: Execution time plot varying the number of threads (Num. tasks = 10).

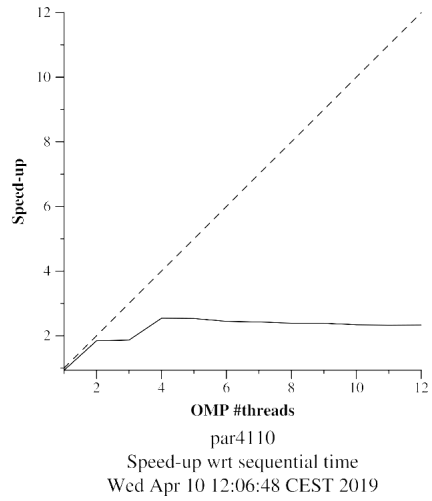


Figure 93: Speedup plot varying the number of threads (Num. tasks = 10).

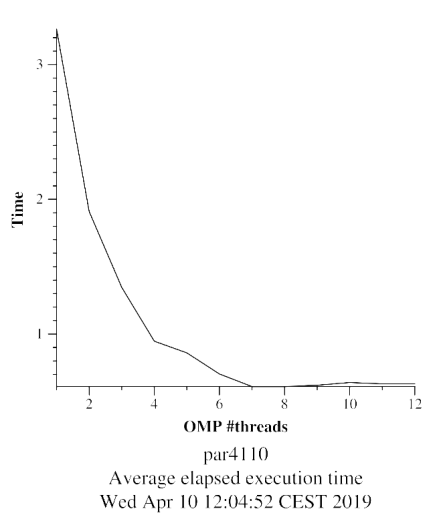


Figure 94: Execution time plot varying the number of threads (Num. tasks = 25).

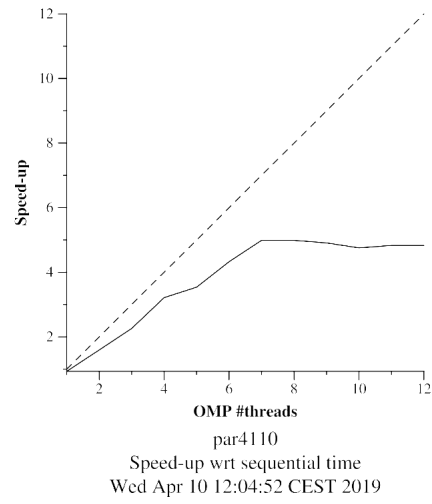


Figure 95: Speedup plot varying the number of threads (Num. tasks = 25).

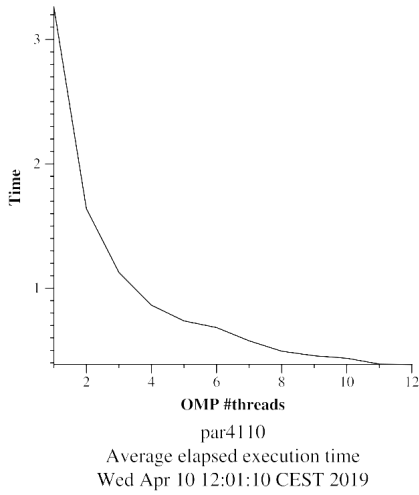


Figure 96: Execution time plot varying the number of threads (Num. tasks = 50).

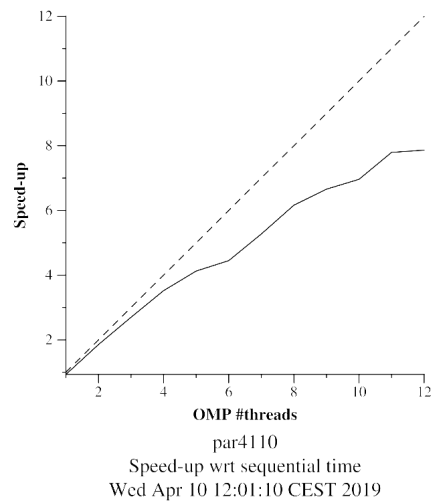


Figure 97: Speedup plot varying the number of threads (Num. tasks = 50).

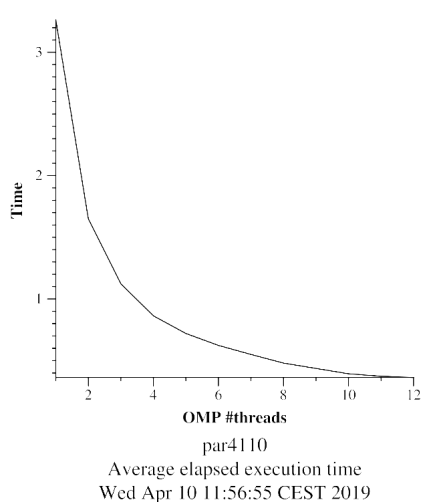


Figure 98: Execution time plot varying the number of threads (Num. tasks = 100).

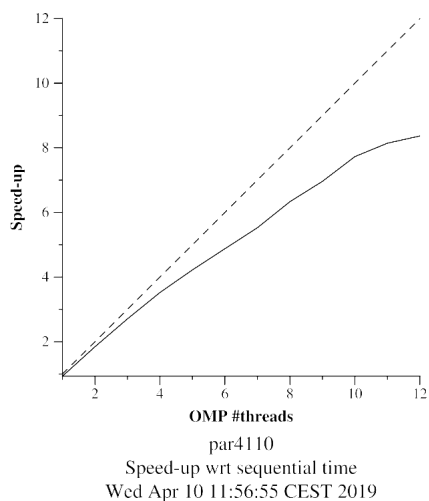


Figure 99: Speedup plot varying the number of threads (Num. tasks = 100).

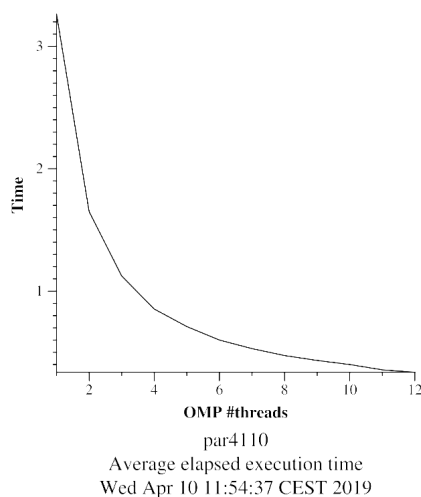


Figure 100: Execution time plot varying the number of threads (Num. tasks = 200).

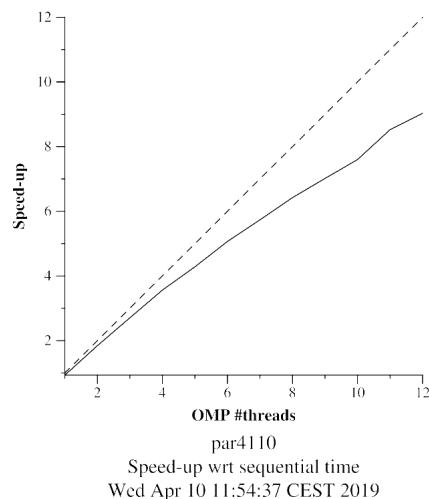


Figure 101: Speedup plot varying the number of threads (Num. tasks = 200).

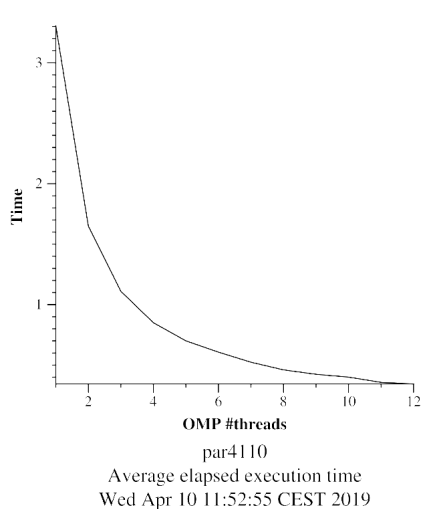


Figure 102: Execution time plot varying the number of threads (Num. tasks = 400).

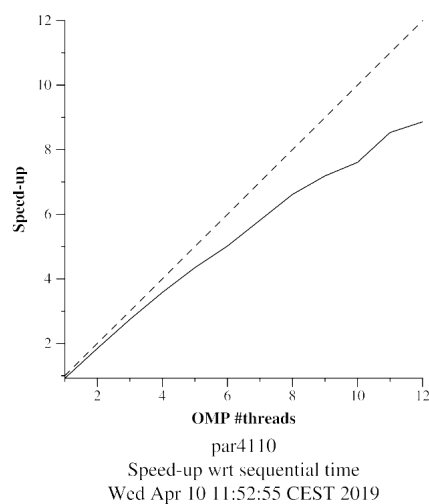


Figure 103: Speedup plot varying the number of threads (Num. tasks = 400).

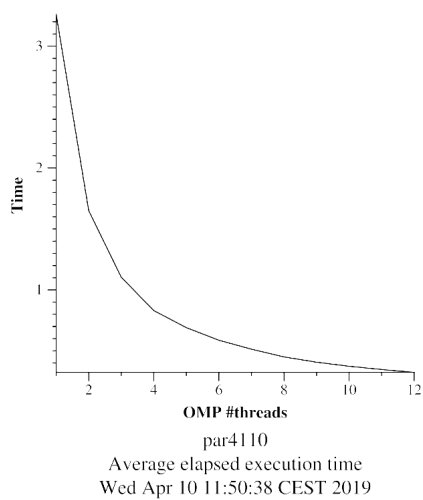


Figure 104: Execution time plot varying the number of threads (Num. tasks = 800).

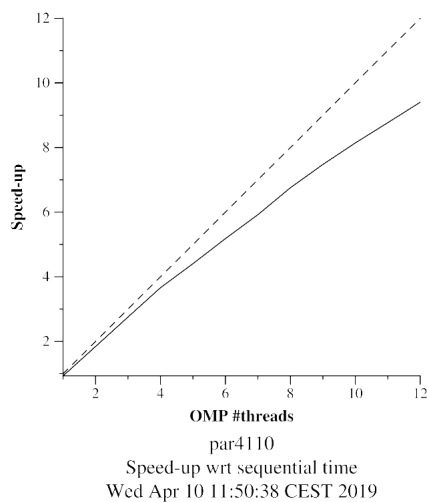


Figure 105: Speedup plot varying the number of threads (Num. tasks = 800).